Algorithms and Data Structures
Module 3. Dynamic programming

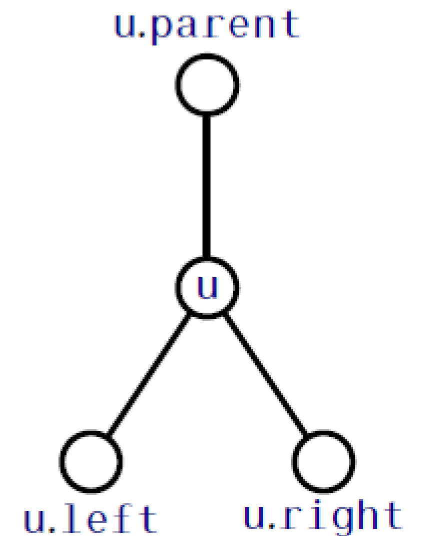
Lecture 17

Optimal binary search tree.

Binary search trees

Binary tree is a graph for which the following conditions hold:

a) It is a tree (=connected acyclic graph).
b) One vertex is marked as the *root* of the tree.
c) Each vertex has 0-2 *children*. Vertices with no children are called *leaves*.
d) For each non-leaf vertex, its children are marked as the *left* child and the *right* child. Even if there is only one child, it is either the left or the right one.



Height of a binary tree is the maximum length of a path from a leaf to the root.

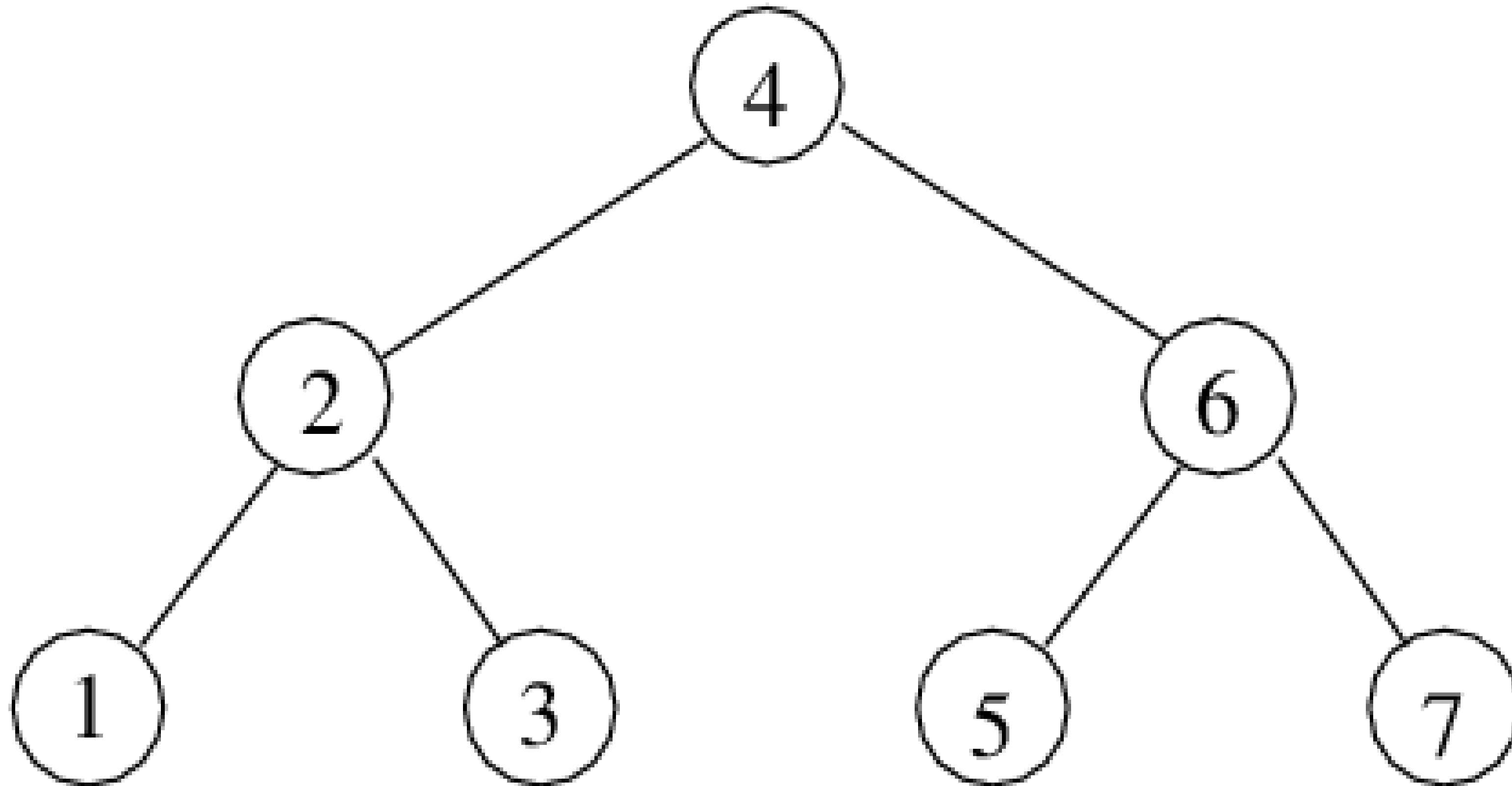
Binary search trees

Binary search tree (*BST*) is a binary tree for which the following conditions hold:

a) Each vertex of BST keeps an item with attached numeric key.

b) BST property holds for each vertex with key *K*:
   - All vertices in the left subtree keep keys which are less than K.
   - All vertices in the right subtree keep keys which are greater than or equal to K.

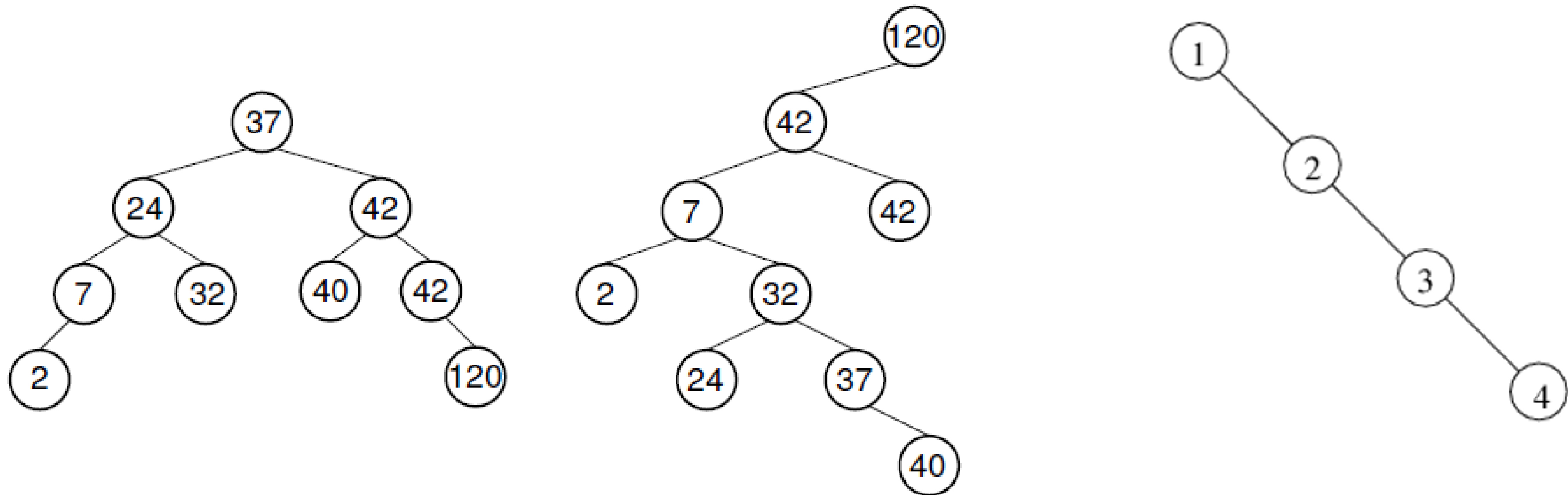
Binary search trees



Binary search trees

Summary of time complexity for BST: `GetMin`, `DelMin`, `Add` have time complexity $O(h)$, where h is the height of the BST.

Height is $O(\log n)$ on average but $O(n)$ in the worst case ☹



Balanced binary search trees

Conclusion: since the complexity of priority queue implementations using trees is $O(h)$, we need balanced trees implementation to achieve $O(\log n)$ worst case complexity for priority queue operations.

A tree is called *balanced* iff its height is $O(\log n)$.

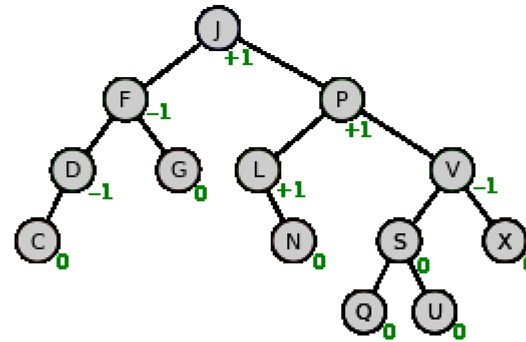
In order to build a balanced tree, we need to

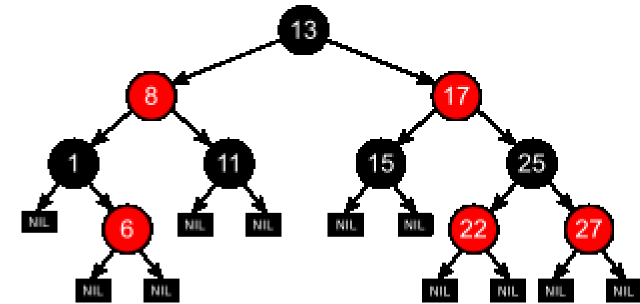
1) Keep additional information (subtrees' heights) for the vertices of the tree.
2) Rebuild (*rebalance*) the tree when it becomes unbalanced after some operations.

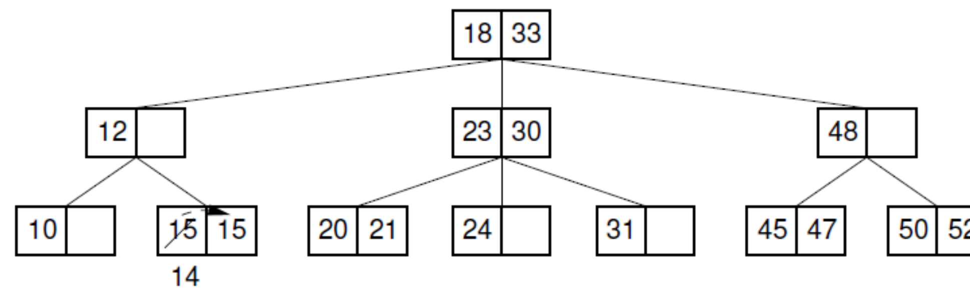
Balanced binary search trees

Balanced trees types:

- AVL trees
- Red-black trees
- 2-3 trees
- …







Optimal binary search trees

A balanced binary search tree is time-optimal for all keys stored in the tree.

There are applications, however, for which different keys have different probabilities / frequencies to be searched.

Examples:

- phonebooks, catalogs;
- dictionaries.

Optimal binary search trees

Let us consider a problem of building an optimal catalog.

1) The catalog must contain a set of objects, whose keys are known in advance: $K = \{k_1, \dots, k_n\}$.
2) The catalog is built before using and is not modified after creation (or modifications are rare compared to search operations).
3) The frequencies (probabilities) of searching are known for all keys: p_i is the probability of searching k_i.

Optimal binary search trees

We want to build a catalog for which the searching time is the least with respect to the given frequencies.

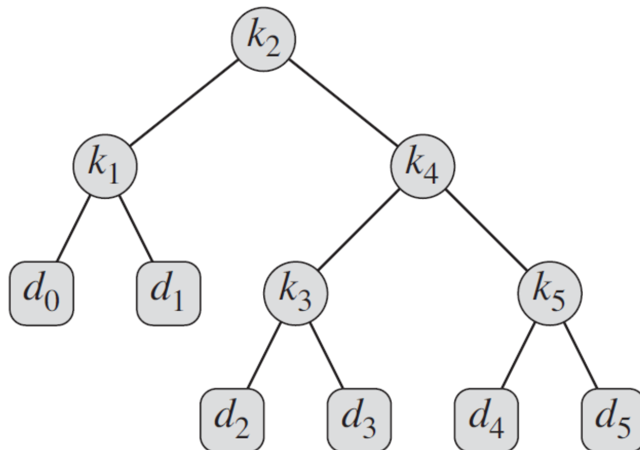
One more issue should be considered: there may be searching for keys absent in K. We suppose that $k_1 < k_2 < \cdots < k_n$ and we know the probabilities q_i $(i = 1, \ldots, n-1)$ of searching values between k_i and k_{i+1}; q_0 is the probability of searching for values less than k_1 and q_n is the probability of searching for values greater than k_n.

Optimal binary search trees

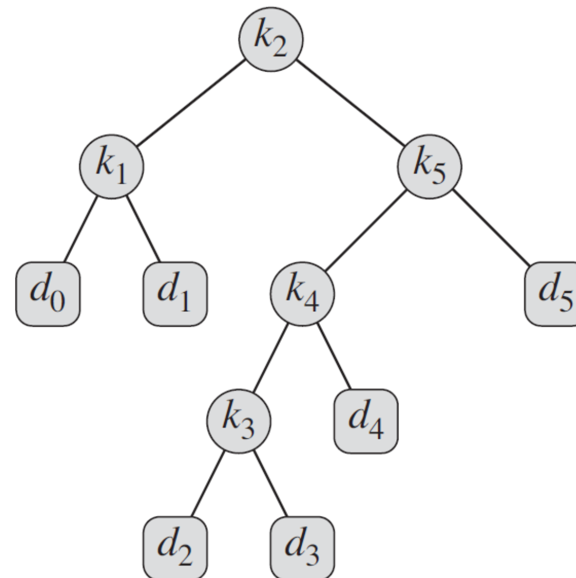
Example (from [Cormen at al. ‘Introduction to algorithms’]): for the given frequencies, tree (a) has expected search cost 2.80 and tree (b) has expected search cost 2.75 and is the optimal BST.

d_i represent ‘dummy keys’.

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10



(a)



(b)

Optimal binary search trees

With these denotations, we have the following condition:

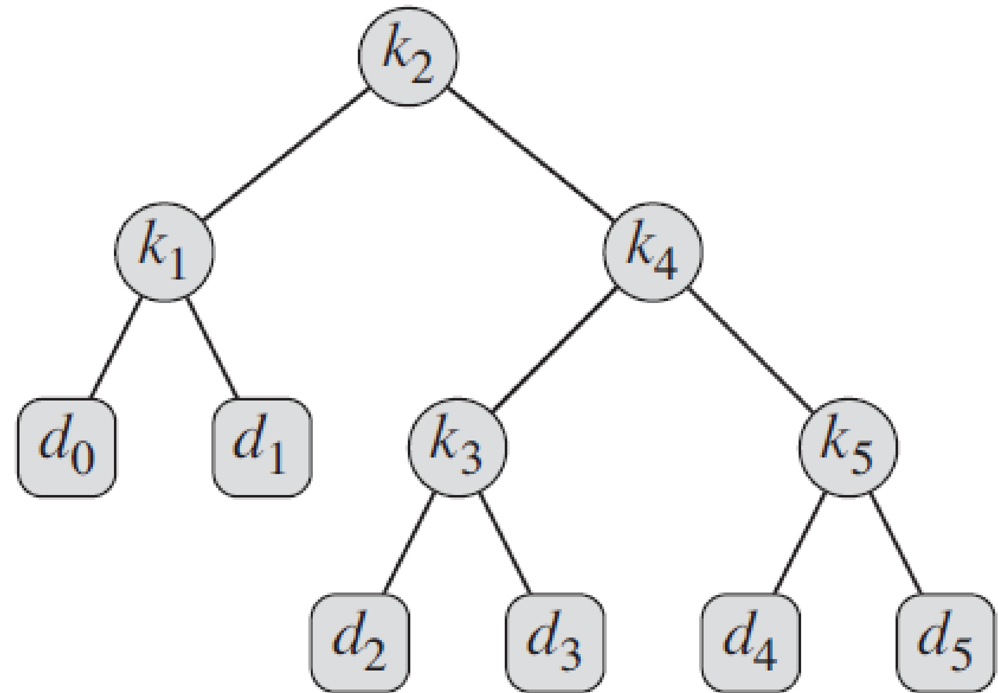
$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

We want to build a tree that minimizes the expected search cost, that is the expected, with respect to the given frequencies, time complexity of a search operation.

Optimal binary search trees

Let us estimate the time complexity of a key k.

- If $k = k_i \in K$, we need $(depth(k_i) + 1)$ comparisons.
- If $k = d_i \notin K$, we need $depth(d_i)$ comparisons.



Optimal binary search trees

Thus, the expected number of comparisons is

$$E(Cost) = \sum_{i=1}^{n} (depth(k_i) + 1)p_i + \sum_{i=0}^{n} depth(d_i)q_i$$

We want to build a tree T that minimizes $E(Cost)$.

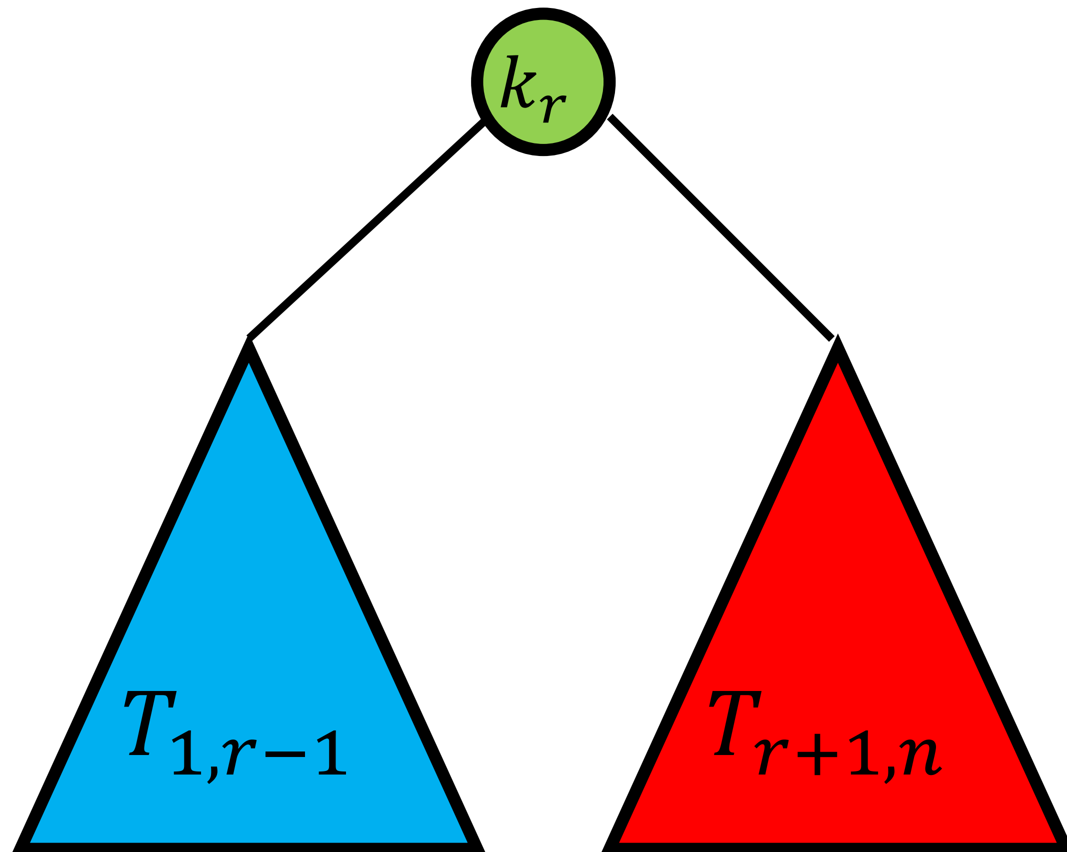
Optimal binary search trees

Let us analyze the structure of an optimal BST $T*$.

Let $T_{i,j}$ denote the optimal BST for keys $\{k_i, \dots, k_j\}$. This means that $T^* = T_{1,n}$. Remember that we suppose that keys are numbered in ascending order: $k_1 < k_2 < \cdots < k_n$.

We can formulate the principle of optimality: if key k_r is at the root of $T_{1,n}$, then the left subtree is $T_{1,r-1}$ (an optimal BST for $\{k_1, \dots, k_{r-1}\}$) and the right subtree is $T_{r+1,n}$ (an optimal BST for $\{k_{r+1}, \dots, k_n\}$)

Optimal binary search trees



Note that for any vertex except the root of depth of the vertex in the tree is its depth on the subtree +1.

Optimal binary search trees

Let us denote $E(Cost)$ for a tree $T_{i,j}$ as $c_{i,j}$.

Also, we denote $w_{i,j} = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$ and call it a 'weight' of the tree. Hence, $w_{i,r-1} + p_r + w_{r+1,j} = w_{i,j}$.

We have the following, for any $i \leq r \leq j$:

$$c_{i,j} = \sum_{l=i}^{j} (depth(k_l) + 1)p_l + \sum_{l=i}^{j} depth(d_l)q_l$$

$$= c_{i,r-1} + w_{i,r-1} + p_r + c_{r+1,j} + w_{r+1,j} = c_{i,r-1} + c_{r+1,j} + w_{i,j}$$

Optimal binary search trees

Thus, we have the recurrence:

$$c_{i,j} = \min_{i \le r \le j} \{c_{i,r-1} + c_{r+1,j} + w_{i,j}\}$$

The base case is $j = i - 1$. In this case $T_{i,j}$ contains one item (d_{i-1}) only, hence $c_{i,i-1} = q_{i-1}$.

$$c_{i,j} = \begin{cases} q_{i-1}, & if\ j = i - 1 \\ \min_{i \le r \le j} \{c_{i,r-1} + c_{r+1,j} + w_{i,j}\}, & if\ i \le j \end{cases}$$

Optimal binary search trees

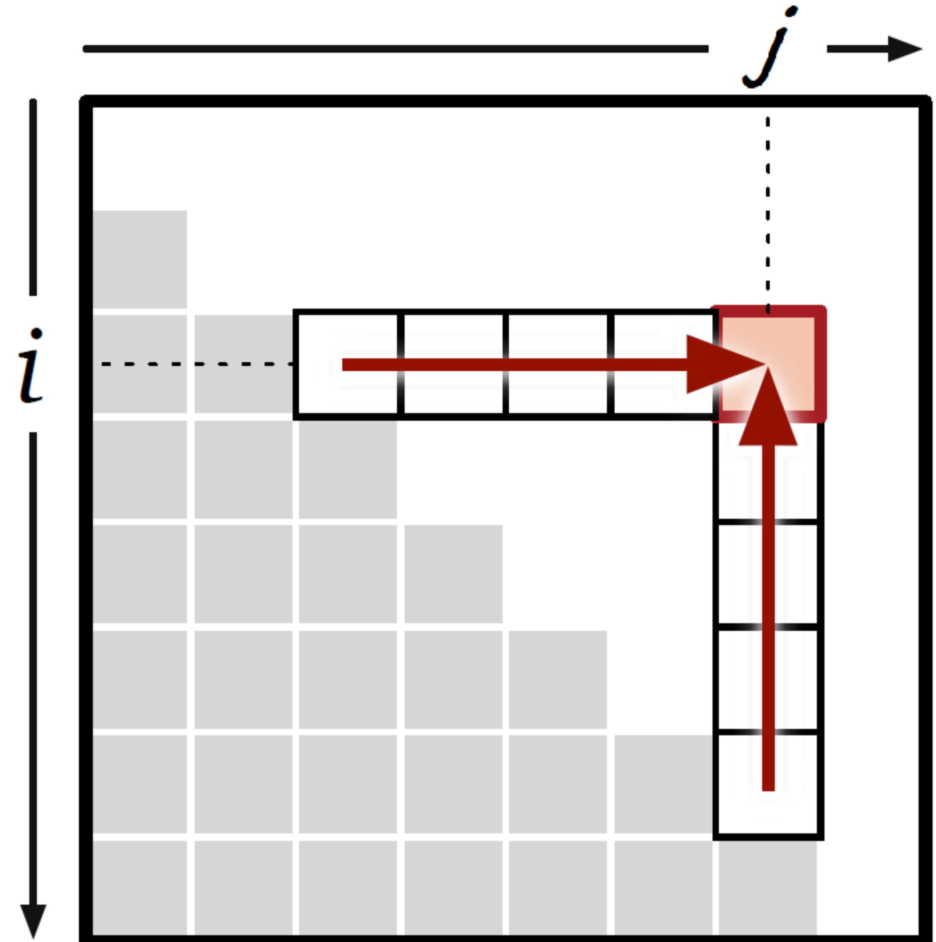
We can implement this recurrence as a dynamic programming procedure.

```
// Create and fill in the helper matrix of weights.
// Actually, we need the right upper triangle
// of this matrix: i ≤ j.
w[1..n, 0..n]
// Create the cost matrix
c[1..n, 0..n]
```

Optimal binary search trees

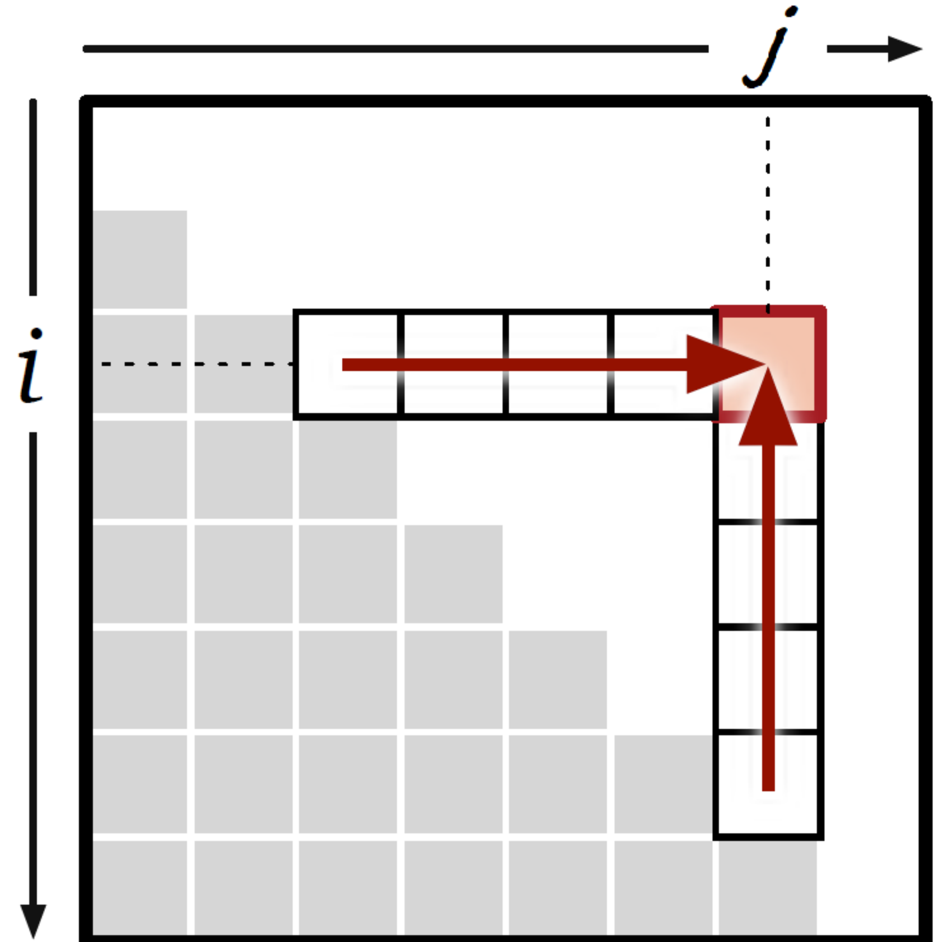
Then, we need to fill in the matrix `c[]`. To calculate `c[i,j]`, we need that values `c[i,t]` and `c[s,j]` be previously calculated, for $t = i + 1, \dots, j - 1$ and $s = i + 1, \dots, j - 1$.

Hence, we need to fill the matrix in an unusual order: diagonally, from the main diagonal to the right upper corner.



Optimal binary search trees

Note that we need to fill the right upper triangle of the matrix.



Optimal binary search trees

It is convenient to organize loops for filling diagonals in the matrix.

```
// Initialize the elements of the starting diagonal
for i=1 to n do: c[i,i-1] = q[i-1];
```

Optimal binary search trees

```
// The outermost loop specifies the number of diagonal
for d=1 to n:
      for i=1 to n:
            j = i+d-1;
            c[i,j] = +∞;
            // find the minimum value
            for r=i to j:
                  tmp = c[i,r-1]+c[r+1,j]+w[i,j];
                  if tmp < c[i,j] then
                        c[i,j] = tmp;
```

Optimal binary search trees

When the matrix is completely filled, `c[1,n]` is the cost of the optimal BST.

To build the optimal BST, we need to store additional information during the calculations.

1. In the initializing step, create matrix `root[1..n,1..n]`.
2. Within the loop for filling the matrix `c[]`, store the optimal value of `r`.

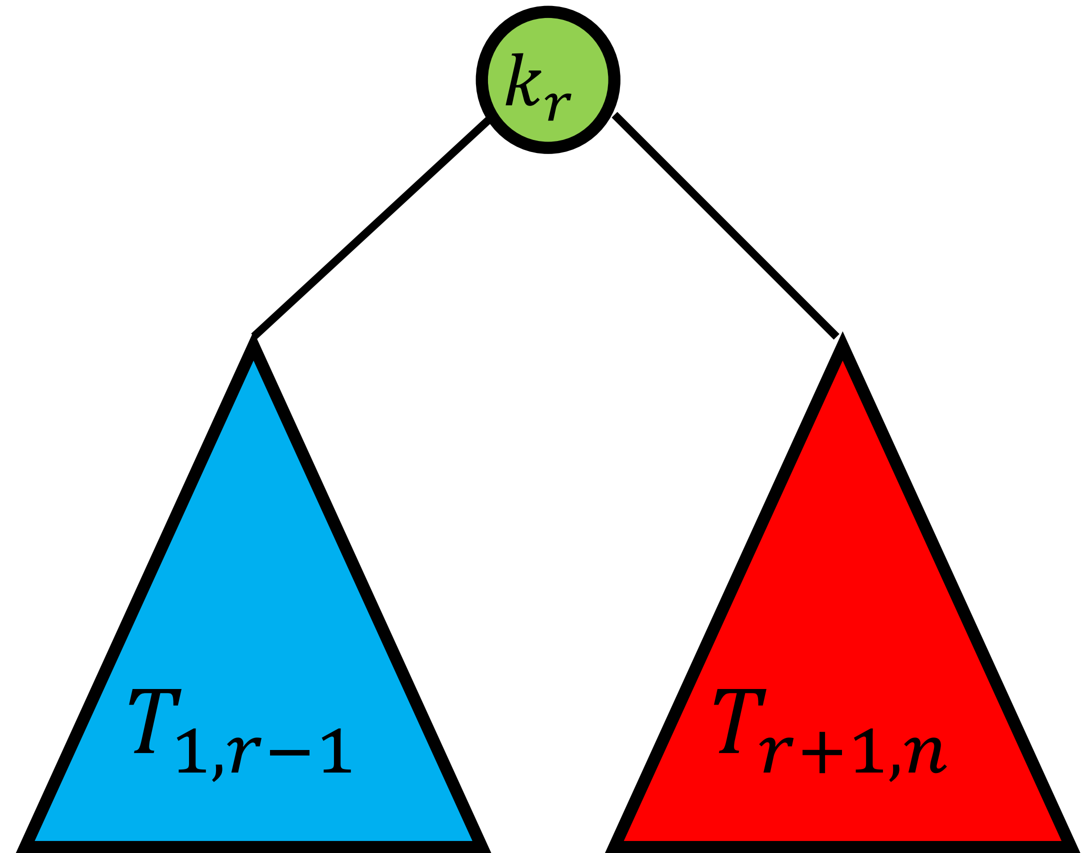
Optimal binary search trees

```
// The outermost loop specifies the number of diagonal
for d=1 to n:
      for i=1 to n:
            j = i+d-1;
            c[i,j] = +∞;
            // find the minimum value
            for r=i to j:
                  tmp = c[i,r-1]+c[r+1,j]+w[i,j];
                  if tmp < c[i,j] then
                        c[i,j] = tmp;
                        root[i,j] = r;
```

Optimal binary search trees

3) Build the optimal BST recursively:

- put `r=root[1,n]` at the root of $T_{1,n}$
- Recursively build optimal BSTs for $\{k_1, \dots, k_{r-1}\}$ and $\{k_{r+1}, \dots, k_n\}$.



Optimal binary search trees

The time complexity of building an optimal binary search tree is $O(n^3)$, since we need to calculate $O(n^2)$ entries and we need $O(n)$ time to calculate each entry.

The space complexity is $O(n^2)$, since we need to store $O(n^2)$ values in matrices `w`, `c`, `root`, for pairs of indices i, j where $j \geq i - 1$.